1.0 TML2 Language Description

1.1 Introduction

The Modular Language 2 (TML2) is an imperative programming language with many influences from Turbo Pascal, C and ADA. Overall syntax and structure is based on Pascal but some constructs have been taken from C and ADA. Procedure call semantics and syntax is taken from C, which means that a procedure identifier is interpreted as a constant pointer and a call is only executed if a parameter list is appended. An empty parameter list is therefor, just like in C, mandatory to call a procedure which has no parameters.

Type declaration syntax is based on Pascal but it has been extended to allow arbitrary complex type expressions in one declaration. This makes it possible to declare complex structured types without naming all sub-types.

One cornerstone of TML2 is modularity which is supported through separately compiled modules (units) based on Turbo Pascal. A module contains an optional interface section where all exported items (constants, types, variables, procedures) are declared and a mandatory implementation section where all private declarations and procedure bodies are defined.

Every module has a possibly empty main program body; a BEGIN-END block in the last part of the module. The main program body of the top module (the one sent to the linker) will be executed first when the linked program is run.

1.2 Reserved words

UNIT, INTERFACE, IMPLEMENTATION, USES, CONST, TYPE, VAR, PROCEDURE, BEGIN, END, IF, THEN, ELSE, ELSIF, FOR, WHILE, DO, LOOP, REPEAT, UNTIL, INC, DEC, INLINE, LENGTH, GENERIC.

1.3 Built in basic types

Type name	Storage size	Description
char	1	Character or unsigned integer value [0..255]
boolean	1	Boolean/logic truth value [true..false]
int	2	Unsigned integer value [0..65535]
long	4	Unsigned integer value [0..4294967295]
pointer	2	Generic pointer or reference to unspecified data.
string	128	Variable length string of characters.

Table 1. *TML2 basic types.*

1.4 Module syntax

```
UNIT <module name> ;
INTERFACE
   <export declarations>
IMPLEMENTATION
   USES <module name> , <module name> ... ;
   <private declarations and procedures>
BEGIN
   <optional main program body>
END <module name> ;
```

The interface section and uses statement are optional and can be omitted.

1.5 Expressions

A TML2 expression is one of the following elements or any valid combination of these:

- constant or literal value
- arithmetic expression
- procedure call with return value

Every expression will, when evaluated, result in a value with an associated type, called the expression type. The expression type is defined by the type of the expression components and the operators or procedures used. Example: the sum of two integer type expressions is also of type integer.

Type checking is based on structural equality and all arithmetic operators require operands of equal types. This means that no automatic type conversion is performed by the compiler. Example to add the integer literal 1 to a variable of type long an explicit type conversion is needed:

```
long_var := long_var + long(1);
```

Arithmetic and logic expressions are constructed in TML2 in the same way as in Pascal. Operator priorities are also the same as in Pascal.

1.6 Condition statements

1.6.1 Simple condition statement

```
IF <condition> THEN
   <statements>
ELSIF <condition> THEN
   <statements>
ELSE
   <statements>
END IF ;
```

The elsif and else parts of a condition statement are optional.

1.6.2 Multiple path condition statement

```
CASE <expression> OF
  <constant> : <statement>
  <constant> : <statement>
  ...
ELSE <statement>
END CASE ;
```

1.7 Iteration statements

1.7.1 Infinite iteration

```
LOOP
  <statements>
END LOOP ;
```

1.7.2 Pre-tested conditional iteration

```
WHILE <condition> LOOP
  <statements>
END LOOP ;
```

1.7.3 Post-tested conditional iteration

```
REPEAT
  <statements>
UNTIL <condition> ;
```

1.8 Other statements

1.8.1 String length operator

The built in operator “length” is defined for variables and expressions of type string. The operator can be used both to get and set the dynamic length of a string. Note that the storage size of a string can not be altered with the length operator.

Example: set length of string s to 12:
length(s) := 12;

Example: get length of string s and assign it to the variable n:
n := length(s);

1.8.2 Increment and decrement operators

A variable can be incremented or decremented by one with the built-in operators inc and dec respectively. Varibles of type char, int, long and pointer are valid.

Example: increment variable “a”, decrement variable “b”:
inc(a);
dec(b);

1.8.3 Explicit type conversion

Unchecked type conversion can be performed by using the target type name as a procedure with one parameter. Conversion to any named type can be performed this way. Simple numeric values are safely type converted by truncation or zero extension. This applies to the basic types: char, int and long. Note that conversion between arbitrary types is done by changing the associated type, data conversion is not performed and the normal compiler consistency checking is disabled. Explicit type conversion may be needed for system programming but it requires that the programmer knows the exact implementation of the converted data types because the normal compiler checking is disabled!

Example: conversion of a character literal to it's corresponding ASCII integer value.
x := int('z');

1.8.4 Inline machine code

To support direct inline machine code there is a built-in special function; inline, which takes an arbitrary no. of parameters and inserts the given values into the compiled code. The inline funtion accepts integer constants, structured constants, variables and procedures as parameters. Inline parameters are interpreted as listed below:

Item	Compiled size	Compiler action
Integer literal	1	Truncated to 8 bits and inserted directly into compiled code.
Integer constant	2	Inserted directly as two 8 bit values, LSB first.
Structured constant	2	Address of constant data taken and inserted as two 8 bit values, LSB first. Automatic relocation.
Global variable	2	Address of variable taken and inserted as two 8 bit values, LSB first. Automatic relocation.
Procedure	2	Address of procedure entry point taken and inserted as two 8 bit values, LSB first. Automatic relocation.

Table 2. *Inline parameter interpretation.*

1.9 Procedures

1.9.1 Procedure declaration syntax

PROCEDURE <name> (<optional parameter list>) <optional return type> ;
  <local declarations>
BEGIN
  <statements>
END <name> ;

Local declarations, statements and procedure name after END are optional. Procedure parameters are declared according to:
VAR <param name> , <param name> ... : <type> ; <param name> ...

The reserved word VAR is optional and is used to declare a parameter as both input and output (parameter passed by reference). Multiple parameters of the same type can be listed with comma separation and the type is defined after a colon. Parameter declarations with different types are separated by semicolons.

Example: procedure declaration:

```
PROCEDURE test(x,y: int; VAR z: char): boolean;
BEGIN
   z := char(x - y);
   return z > ' ';
END test;
```

1.9.2 External procedures at absolute addresses

Interfaces to external code at absolute addresses can be created by declaring TML2 procedure prototypes at absolute addresses. Such a procedure can later be called as usual and the external code address will be called. This feature allows mixing TML2 with legacy assembly code, external code in PROM etc. Note that the external call will follow TML2 calling conventions: parameters are pushed in reverse order (first parameter is pushed last), parameters are removed by caller and the frame pointer (IY) register must be preserved.

Example: external procedure declaration:

```
PROCEDURE get_env(key: string): ^string; EXTERN $20;
```

1.9.3 Variable length parameter lists

A procedure with type safe variable length parameter lists can be declared by using the reserved word “generic” in the parameter list declaration. All built-in basic types in any number and order are compatible with a generic procedure parameter. Inside the procedure body a generic parameter is accessed as a variable length array with a structure representing the actual parameter for each index. The structures has the fields “typecode” and “value”. The actual no. of parameters can be obtained as the array length:
no_of_args := length(generic);

The first parameter is stored in generic[1]. A simple form of runtime type information is used, the type of each actual parameter is stored as an integer value in the corresponding “typecode” field. Parameter values are stored as integers in the “value” field but should be interpreted as described below.

Param type	Typecode	Interpretation of value
char	1	Character value is stored in 8 LSb.

Table 3. *Runtime type encoding for generic parameters.*

Param type	Typecode	Interpretation of value
int	2	Integer value is stored directly.
boolean	3	Boolean value is stored in 8 LSb.
pointer	4	Pointer value is stored directly.
string	5	Address of string is stored directly.

Table 3. *Runtime type encoding for generic parameters.*

Example: procedure with variable length (generic) parameter list.

```
PROCEDURE test(generic);
 var
    i : int;
BEGIN
    for i := 1 while i <= length(generic) do inc(i) loop
        do_something(generic[i].typecode,generic[i].value);
    end loop;
END test;
```

1.10 Forward declaration and mutual recursion

Mutual recursion and circular call dependencies are handled by forward declaration of procedures. A forward declaration declares a procedure call signature (procedure prototype) without implementing the body. Every forward declared procedure must later be implemented with a complete declaration in the same compilation unit. Procedures declared in the unit interface section is handled like a forward declared procedures.

Example: mutual recursion and forward declaration.

```
PROCEDURE x(); forward;

PROCEDURE y();
BEGIN
    x();
END;

PROCEDURE x();
BEGIN
    y();
END;
```

2.0 Compiler tools

2.1 Overview

To transform a TML2 source program into an executable program and in some cases into a form suitable for storage and execution in ROM a collection of software tools are needed. These tools are involved:

- Compiler is used to transform a source module into a binary object file.
- Linker is used to combine one or more object files into a single executable file containing relocatable machine code.
- Loader is used to process an executable file into a raw binary image file with absolute addresses. This is only needed for ROM-applications.
- Dump utility may be used for inspection of binary object files. File contents are converted into a human readable form.

2.2 Compiler - tml2

The compiler is a self-contained program. It is stored in the file tml2.exe (DOS/ Windows) or tml2 (Unix).

Command line syntax:

tml2 [options] source-file

Source file name can be given with or without .tml suffix but the compiler will append the .tml suffix if missing. After a successful compilation an object file with the source base name and .o suffix will be generated.

Examples:

tml2 test
The source file “test.tml” is read and “test.o” is written.

tml2 test.tml
The source file “test.tml” is read and “test.o” is written.

Option	Description
-g	Generate and store debug info in object file. Line number info and local variable symbols are stored.
-s	Suppress loading of runtime support library (support.o).

Table 4. *TML2 compiler options*

Option	Description
-d	Enable compiler internal debug mode. Will write detailed internal trace information on stdout (console). Intermediate code etc is also written to a <name>.tr file, where <name> is the source file base name.
-O0	Compile without any optimization.
-O1	Optimize code size.
-O2	Optimize for execution speed. Some runtime support functions are expanded inline.
-ovl	Compile complete unit for code overlay. String literals in unit are also overlaid.
-ovl_fn	Enable per-function selectable code overlay. Pragma statements in unit source code are used to select which functions will be overlaid. [pragma overlay | pragma no_overlay]
-Lpath	Setup search path for compiler to find referenced object files. Multiple file paths can be given, separated by semicolons ";". Example: tml2 -L..\common;c:\src\lib;d:\ test
-lstat	Allocate all local variables in data segment, not on stack. This option may increase execution speed and decrease code size. NOTE: recursion, interrupts or multi-threading is usually not compatible with this option.

Table 4. *TML2 compiler options*

2.3 Linker - link

The linker is a self-contained program. It is stored in the file link.exe (DOS/ Windows) or link (Unix). Linkage begins with reading the given main unit object file and then resolving all external references by loading other object files as needed. When all references are resolved a singe executable file with all combined code and data sections are written. A .sym text file is also generated with all global symbols and there assigned relative addresses are listed. The .sym file name is also based on the main base name.

Command line syntax:

link [options] main-object-file

Object file name can be given with or without .o suffix but the linker will append the .o suffix if missing. After a successful linkage an executable file with the main base name without suffix will be generated.

Option	Description
-g	Combine debug info from all input files and store in output file.
-d	Enable linker internal debug mode. Will write detailed internal trace information on stdout (console).
-Lpath	Setup search path for linker to find referenced object files. Multiple file paths can be given, separated by semicolons ";". Example: link -L..\common;c:\src\lib;d:\ test
-v1	Write output file in version 1 executable format, compatible with earlier versions of OS-X. Output file will be given .exe suffix. NOTE: overlays are not supported with this option enabled.

Table 5. *TML2 linker options*

2.4 Loader - ld

The loader is a self-contained program. It is stored in the file ld.exe (DOS/ Windows) or ld (Unix).

Command line syntax:

ld [options] executable-file [address | code-address data-address]

Executable file name must be given in full, usually without any suffix. After successful loading a binary image file with the main base name and .com suffix will be generated. The image file contains only code with absolute addresses. It is suitable for execution in ROM.

Base address for loading can be given as one or two base addresses. One single base address for code with data area appended. Useful for execution in RAM.

Or two addresses, first the code base address then the data area base address. This is needed for a system with code in ROM and data area in RAM. Addresses can be given as decimal numbers or hexadecimal with a $ prefix.

Examples:

ld my_program 4096
Create image for execution in RAM at address 4096.

ld my_program $100 $C000
Create image for execution at address 256 (100 hex) and data area at address 49152 (C000 hex).

Option	Description
-d	Enable linker internal debug mode. Will write detailed internal trace information on stdout (console).

Table 6. *TML2 loader options*

2.5 Dump utility - tmldump

The dump utility is a self-contained program. It is stored in the file tmldump.exe (DOS/Windows) or tmldump (Unix). It is used to display the contents of a TML2 object file or executable file.

Command line syntax:

tmldump [options] object-file

Option	Description
-d	Enable linker internal debug mode. Will write detailed internal trace information on stdout (console).
-exe	Read executable file, assume correct magic number and structure for executable file instead of object file.

Table 7. *TML2 dump utility options*

3.0 Formal syntax

3.1 Notation used

Terminal symbols are written in uppercase letters, non-terminal symbols are written as names within angular braces, e.g. <non-terminal>
Curly braces, vertical bar and right-brace-star are meta symbols. Some examples:

Optional construct:	{ optional }
Optional construct with 0..N repeats:	{ optional-repeatable }*
Alternative constructs:	alt-a | alt-b | alt-c

3.2 Module

```
<program> ::=       UNIT <id> ;
                    { INTERFACE <interface_declaration> }
                    IMPLEMENTATION
                    { USES <id_list> ; }
                    { <data_declaration> | <func_declaration> }*
                    <body_declaration>

<interface_declaration> ::= { <data_declaration> | <func_header> }*
```

3.3 Declarations

```
<data_declaration> ::=          VAR <var_declaration> |
                                CONST <const_declaration> |
                                TYPE <type_declaration>

<var_declaration> ::=           <id_list> : <type_expr> ;

<const_declaration> ::=         <id> = <literal> ; |
                                <id> : <type_expr> = ( <const_list> ) ;

<type_declaration> ::=          <id> = <type_expr> ;

<func_header> ::=               PROCEDURE <id> <func_expr> ;

<func_declaration>  ::=         <func_header> FORWARD ; |
                                <func_header> EXTERN <int> ; |
                                <func_header> { <data_declaration> }*
                                        <body_declaration>

<const_list> ::=        <constant> { <constant> }*

<constant> ::=          <id> | <scalar_literal>

<id_list> ::= <id> { , <id> }*

<type_expr> ::=         <id> |
                        ^ <type_expr> |
                        ARRAY [ <expr> ] OF <type_expr> |
                         RECORD { <id> : <type_expr> ; }* END |
                         FUNCTION <func_expr>

<func_expr> ::=         { ( <arg_declaration> ) } { : <type_expr> }

<arg_declaration> ::=           <var_declaration> { <arg_declaration> }* |
                                VAR <var_declaration> { <arg_declaration> }*


<body_declaration> ::= BEGIN <stmt_list> END { <id> } ;
```

3.4 Statements

<stmt_list> ::= { <stmt> }*

<stmt> ::= <if_stmt> |
<while_stmt> |
<loop_stmt> |
<repeat_stmt> |
<case_stmt> |
<return_stmt> |
<length_stmt> |
<code_stmt> |
<expr_stmt>

<if_stmt> ::= IF <expr> THEN <stmt_list> { ELSE <stmt_list> } END { IF } ;

<while_stmt> ::= WHILE <expr> LOOP <stmt_list> END { LOOP } ;

<loop_stmt> ::= LOOP <stmt_list> END { LOOP } ;

<repeat_stmt> ::= REPEAT <stmt_list> UNTIL <expr> ;

<case_stmt> ::= CASE <expr> OF { <case_branch> }* { ELSE <stmt> } END { CASE } ;

<case_branch> ::= <int> : <stmt> |
<char> : <stmt>

<return_stmt> ::= RETURN { <expr> } ;

<length_stmt> ::= LENGTH (<expr>) := <expr> ;

<code_stmt> ::= CODE ({ <code_list> }) ;

<code_list> ::= <id> { , <code_list> }* | <int> { , <code_list> }*

3.5 Expressions

<expr_stmt> ::= <expr> { := <expr> } ;

<expr> ::= <sExpr> <expr'>

<expr'> ::= { <relop> <sExpr> <expr'> }

<sExpr> ::= <term> <sExpr'>

<sExpr'> ::= { <addop> <term> <sExpr'> }

<term> ::= <factor> <term'>

<term'> ::= { <mulop> <factor> <term'> }

<factor> ::= LENGTH (<designator>) |
(<expr>) |
- <factor> |
NOT <factor> |
<int> |
<char> |
<string> |
<designator>

<designator> ::= <id> <qualifier>

<qualifier> ::= { <aggregate> | <index> | . <id> | ^ }*

<aggregate> ::= ({ <expr> { , <expr> }* })

<index> ::= [<expr>]

<relop> ::= < | > | <= | >= | = | <>

<mulop> ::= * | / | MOD | AND

<addop> ::= + | - | OR

3.6 Primitive non-terminal symbols

<id> ::= <letter> { <letter> | <digit> }*

<scalar_literal> ::= <char> | <int>

<int> ::= <digit> { <digit> }* |
$ <hexDigit> { <hexDigit> }*

<char> ::= # <int> | ’<character> ’

<string> ::= " { <character> }* "

<letter> ::= A | B | C ... | Z | a | b .. | z | _

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<hexDigit> ::= <digit> | A | B | C | D | E | F

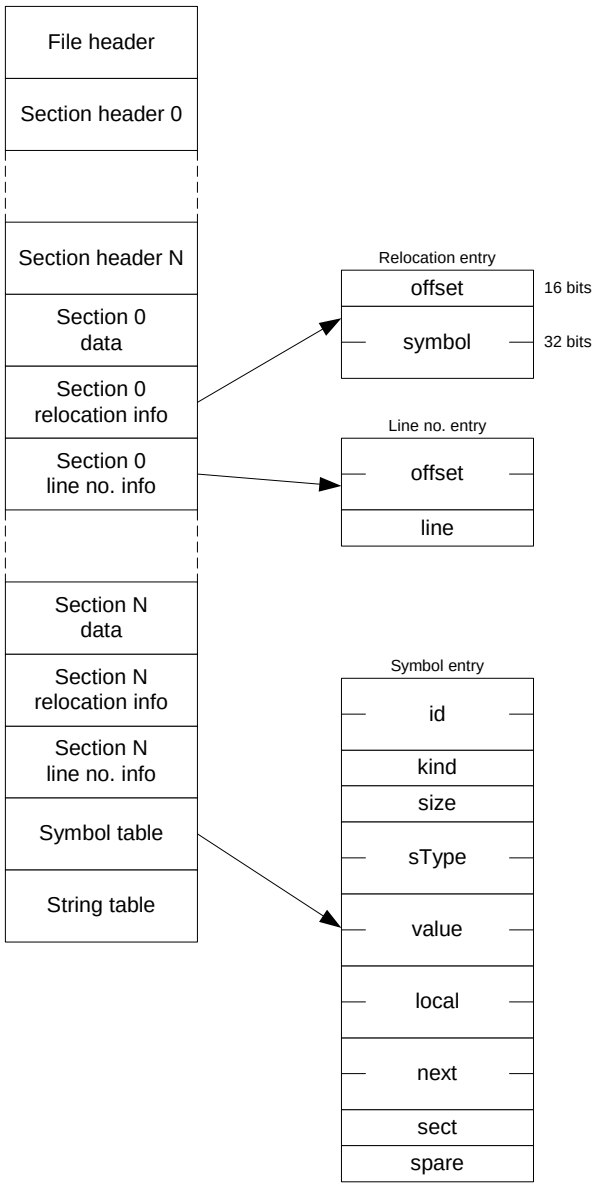
<character> ::= <any 8 bit ASCII character>

4.0 TML2 Compiler internals

4.1 Object file format

An object file has the extension .o and contains relocatable code, constant data and symbol table for a compiled TML2 module. An object file is a structured binary file. An executable file usually has the extension .exe and has the same structure as an object file. The difference is that an executable file is linked and does not contain any external references. A symbol table is optional in an executable file.

4.1.1 Basic structure



Object file contents are described in the table below.

Part	Description
file header	File type, file format version, no. of sections and file offsets for global parts etc are stored here.
section header	Section type, name, size and file offsets for section data, relocation and line number info are stored here. There can be 0-N section headers in an object file.
section data	Raw data for section is stored here. Compiled code, initialized data etc. Un-initialized data sections are not stored in file.
relocation info	All internal and external references are stored here.
line no. info	Optional line number debug info is stored here.
symbol table	Internal and referenced external symbols are stored here. Symbol identifiers are stored as offsets for the string table.
string table	Identifier string data is stored here. Stored as TML2 strings, each with a leading 8 bit string length field.

Table 8. *TML2 object file contents.*

4.1.2 Object file header format

The TML2 object file header is structured as shown below.

Offset	Size	Description
0	2	File type, 17=object file, 59=executable file
2	8	Compiler name, text string, usually TML2.
10	2	Compiler version, 2 = TML2
12	16	Name of source module, text string.
28	2	Compilation version number, incremented by compiler.
30	2	Object file checksum, all bytes summed with 16 bits modulus.
32	2	Code segment offset in object file.
34	2	Size of code segment in bytes.
36	2	Size of data segment in bytes. Actual data segment is not stored in object file.
38	2	Size of heap segment in bytes. Actual heap segment is not stored in object file.
40	2	Size of stack segment in bytes. Actual stack segment is not stored in object file.
42	2	Code relocation table offset in object file.
44	2	Number of code relocation records.
46	2	Data relocation table offset in object file.
48	2	Number of data relocation records.
50	2	Symbol table offset in object file.
52	2	Number of symbol records.
54	2	String table offset in object file.
56	2	Size of string table in bytes.
58	2	Debug info offset in object file.
60	2	Number of debug records.

Table 9. *TML2 object file header format.*

4.1.3 Object file relocation record format

Code relocation records are structured as shown below.

Offset	Size	Description
0	2	Offset in section data for 16 bit address which is a reference to a symbol. Address should be relocated relative to actual section base address.
2	4	Index of referenced symbol in symbol table. 0 = no symbol.

Table 10. *Code relocation record format.*

4.1.4 Object file line number entry format

Line number entries are structured as shown below.

Offset	Size	Description
0	4	Offset in section data for start of given line in associated source code file.
2	2	Line number.

Table 11. *Line number entry format.*

4.1.5 Object file symbol table format

Symbol table records are structured as shown below.

Offset	Size	Description
0	1	Symbol type, 0=code, 1=data, 2=type, 3=class, 4=compilation unit.
1	1	Symbol location, 0=absolute, 1=internal, 2=external, 3=other.
2	2	Identifier, stored as string table offset.
4	2	Associated symbol 1, stored as symbol table index.
6	2	Associated symbol 2, stored as symbol table index.
8	2	Address, offset or value. Depends on type of symbol.
10	2	Size of referenced object in bytes.
12	2	Reserved

Table 12. *Symbol table format.*